Safety of 64-bit code

Andrey Karpov
OOO "Program Verification Systems"

August 2009

Abstract

The article reviews the issues of providing safety of program code when adapting it for 64-bit systems.

Introduction

We will not speak about a threat of your software being cracked and about extent of damage caused in this case. There are many books and articles devoted to this topic. So let's pass on to a new practical issue in the sphere of increasing program code's safety relating to mastering 64-bit systems. I think you will not be astonished that we will speak about C/C++ languages for which safety issues are especially crucial.

Because of errors and defects, program code can become more subject to attacks using buffers' overflow when being ported from 32-bit systems on 64-bit ones. It relates to change of base data types what can be used to attack the code. In other words, the code which was safe in a 32-bit system and could not be used for a break-in, can become unsafe after being recompiled for 64-bit systems.

The problem of 64-bit code's safety is not a new aspect in the sphere of information security. The problems of different behavior of code and probability of its being cracked have always depended on the hardware platform being used. But mass migration to 64-bit systems urges us to single out the tasks of 64-bit code's safety into a separate category which demands close attention and individual research. In this article, we will try to touch upon the problems of 64-bit code's safety and draw attention of the developers of software and security systems to this new source of potential danger when developing modern 64-bit solutions.

Analysis of program code

There are various approaches to program code security. We will take into account static code analysis for it is the most suitable method for the task of searching defects when porting code on another platform.

There are a lot of various static analysis tools providing diagnosis of potentially unsafe code sections which can be used for various types of attacks. For example: ITS4, SourceScope, Flawfinder, AK-BC.

By the way, I have learned an interesting thing recently. I have always considered static analysis tools to be tools of searching errors in programs with the purpose to make it safer and more stable to input data. But it turned out that hackers also use static analysis tools but with quite an opposite purpose [1]. They detect potentially unsafe sections in programs to examine them in detail further. It is nearly impossible to

look through the code of modern applications because of their sizes, so static analysis is a good help. After disassembling the code hackers sieve the most interesting code sections for further examination with the help of static analysis. For example, they can search the code which uses line copying and at the same time contains increase/decrease of register or a memory cell in one. Programmers make mistakes very often while working with lines when they have to reserve an additional byte for the terminal symbol 0x00 (end of line). This code usually contains magic arithmetic combinations which have -1 or +1. And of course a code like this is interesting for a hacker because he can perform an attack using buffer overflow.

But we have digressed. Static analyzers help programmers detect potentially unsafe code sections in their programs and one should not underestimate their help. Let's consider some examples of code which becomes unsafe or even incorrect after being ported on a 64-bit system.

Examples of incorrect and vulnerable code

You can learn about many errors occurring in 64-bit programs in the articles "20 issues of porting C++ code on the 64-bit platform" [2] and "Some examples of the 64-bit code errors" [3]. But in these articles those errors are emphasized which cause failure of a program but not from the viewpoint of its being vulnerable to attacks.

Unfortunately, the author failed to find systematic works on the issues of providing security of 64-bit code. And it seems that vulnerability patterns specific of 64-bit systems are a new task to be investigated. Still, let's try to examine some examples.

One of attack methods is transfer of a large data size exceeding, for example, 4 Gb into a program.

```
void *SpecificMalloc(unsigned int size) {
  return malloc(size);
}
...
char *buf;
size_t len;
read(fd, &len, sizeof(len));
buf = SpecificMalloc(len);
read(fd, buf, len);
```

We would remind you that in 64-bit systems (Linux, Windows), int type's size is 32 bits while the size of size_t is 64 bits. The error consists in converting size_t type to unsigned int type when calling SpecificMalloc function. If the size of the file is more than 4 Gb the array's limits will be overrun while reading data and this is an error. Of course, the error is evident in this example, but it shows how dangerous explicit and implicit type conversions can be occurring in a 64-bit code which uses 32-bit and 64-bit types together to store sizes, indexes etc.

Another type of threats is using fixed sizes of buffers and magic constants. Especially it is relevant to old code written about ten years ago by the programmers who did not think that the size of a pointer or variable of time_t type would change sometime.

Let's consider a simple example of an overflow of a buffer with a fixed size:

```
char buf[9];
sprintf(buf, "%p", pointer);
```

You can see this in programs. Especially in old ones.

Let's consider another example where using magic number 4 causes an error of necessary memory size allocation:

```
LPARAM *CopyParamList(LPARAM *source, size_t n)
{
  LPARAM *ptr = (LPARAM *)malloc(n * 4);
  if (ptr)
    memcpy(ptr, source, n * sizeof(LPARAM);
  return ptr;
}
```

Program operation logic can also change unexpectedly:

```
int a = -2;
unsigned b = 1;
ptrdiff_t c = a + b;
if (c == -1)
{
  printf("Case: 32-bit\n");
} else {
  printf("Case: 64-bit\n");
}
```

In this inaccurate code different branches of 'if' operator will be executed depending on the platform's capacity. According to C++ rules "ptrdiff_t c = a + b;" expression will be evaluated in the following way:

- The value of int type -2 will be converted into unsigned type with the value 0xFFFFFFFEu.
- Two 32-bit values 0x00000001u and 0xFFFFFFFEu will be summed and we will get a 32-bit number 0xFFFFFFFFu.
- 0xFFFFFFFFu value will be placed into a 64-bit variable of signed type. In case of a 32-bit system it means that the variable will contain value -1. In case of a 64-bit system it still will be 0xFFFFFFFF.

Such effects are dangerous not only in logical expression but when working with arrays as well. A particular combination of data in the following example causes writing outside the array's limits in a 64-bit system:

```
int A = -2;
unsigned B = 1;
int array[5] = { 1, 2, 3, 4, 5 };
int *ptr = array + 3;
ptr = ptr + (A + B);
*ptr = 10; // Memory access outside the array
           // in case of 64-bit environment.
```

Such an error can be used if you were lucky to detect the incorrect value of A and B variables so that you could write the data into a memory section you need.

Errors in program logic can easily occur in the code processing separate bits. The next type of errors relates to shift operations. Let's consider an example:

```
ptrdiff_t SetBitN(ptrdiff_t value, unsigned bitNum) {
  ptrdiff_t mask = 1 << bitNum;
  return value | mask;
}
```

This code operates correctly on a 32-bit architecture and allows you to set the bit with numbers from 0 to

31 into one. After porting the program on a 64-bit platform you have to deal with setting bits with numbers from 0 to 63. But this code will never set the bits with the numbers 32-63. Pay attention that "1" has int type and an overflow will occur after the shift in 32 positions. As a result we will get 0 or 1 depending on the implementation of the compiler. Note also that incorrect code will cause one more interesting error. After setting the 31st bit in a 64-bit system the result of the function's operation will be the value 0xffffffff80000000. The result of "1 << 31" expression is the negative number -2147483648. This number is represented in a 64-bit variable as 0xffffffff80000000.

If one manipulates the input data of such incorrect errors one can get illegal access, if, for example, access rights masks defined by separate bits are processed.

If the examples given above seem to you farfetched and imaginary, I advise you to get acquainted with one more code (in a simpler form) which has been used in a real application in UNDO/REDO subsystem, although it seems very strange:

```
// Here the indexes are saved as a line
int *p1, *p2;
....
char str[128];
sprintf(str, "%X %X", p1, p2);

// And in another function this line
// was processed in this way:
void foo(char *str)
{
  int *p1, *p2;
  sscanf(str, "%X %X", &p1, &p2);
  // As a result we have an incorrect value of p1 and p2 pointers.
  ...
}
```

The result of manipulations with the pointers using %X is incorrect behavior of the program in a 64-bit system. This example does not only show the 64-bit code's security problem but also how dangerous the depths of large and complex projects which have been written for many years are. If a project is rather large and old it is likely that it contains defects and errors relating to supposition about the sizes of different data structures, data alignment rules and etc.

Diagnosis of vulnerabilities in 64-bit code

Let's first systematize the types of goals which become subject to attacks after porting code on a 64-bit system:

1. Code sections with arithmetic expressions in which 32-bit and 64-bit data types are used together are dangerous.
2. Code sections with address arithmetic containing operations with 32-bit data types are dangerous.
3. We should pay attention to expressions containing magic constants which can denote data types' sizes, maximum acceptable values and data shifts in data structures.
4. The code containing shift operators or other bit operations may become a goal for an attack.
5. Various operations of explicit and implicit conversion of 32-bit and 64-bit types can be a potential threat.
6. The code implementing reading or writing of data which contain types changing their sizes on a 64-bit system is also dangerous.

This list cannot be called full yet for it is, actually, one of the first investigation articles on the topic of

safety of code being ported on a 64-bit system. But verification of even these objects can help increase code safety and remove both many vulnerabilities and errors which can occur even at correct data.

At the moment, there is no separate product for controlling safety of code when porting it on 64-bit systems. But we have PVS-Studio static code analyzer which fully supports diagnosis of all the problems relating to 64-bit vulnerabilities described in this article.

PVS-Studio program product is a development by the Russian company OOO "Program Verification Systems" and is intended for verifying modern applications using such technologies as parallel programming (OpenMP) and 64-bit architectures [4]. PVS-Studio integrates into Microsoft Visual Studio 2005/2008 environment and into MSDN Help system as well.

Viva64 subsystem included into PVS-Studio helps a specialist track in the source code of C/C++ programs potentially unsafe fragments relating to porting software from 32-bit systems on 64-bit ones. The analyzer helps write safe correct and optimized code for 64-bit systems.

Abilities of PVS-Studio cover diagnosis of vulnerability problems in 64-bit program code described above. Diagnostic abilities of this analyzer are more than enough for solving only tasks of providing security of 64-bit code because it is intended not only for detecting potential errors but for search of non-optimal data structures as well. However, you can switch off any unnecessary warnings with the help of settings.

I would like you to note that PVS-Studio is intended for detecting errors occurring when porting 32-bit programs on 64-bit systems or when developing new 64-bit programs. But PVS-Studio cannot diagnose errors which may occur when using functions dangerous on any platforms such as sprintf, strncpy and so on. To diagnose such errors you must use the tools we have mentioned - ITS4, SourceScope, Flawfinder, AK-BC. PVS-Studio supplements these tools bridging the gap in the sphere of diagnosing 64-bit problems but does not replace them.

Conclusion

While being involved into the process of providing security, never give preference only to one sphere being it static or dynamic analysis, testing at incorrect input data etc. Safety of a system is determined by its weakest point. It can happen that a system's safety can be increased in many times with the help of a simple administration method, for example, a lock.

There is a legend which may be true that once during security audit in some company it was assigned the worst mark, even before the specialists began to check if the data had been copied, what software had been installed on the server and so on. Well, the server was situated in some room with a non-lockable door and any one could enter it. Why? It had been too noisy, so they put it far from the offices so that it did not disturb the workers.

References

1. Greg Hoglund, Gary Mcgraw. Exploiting Software: How To Break Code. Publisher: Addison-wesley Professional. ISBN: 0201786958
2. Andrey Karpov, Evgeniy Ryzhkov. 20 issues of porting C++ code on the 64-bit platform. http://www.viva64.com/art-1-2-599168895.html
3. Evgeniy Ryzhkov. Some examples of the 64-bit code errors. http://www.viva64.com/articles/PortSample.html
4. Evgeniy Ryzhkov. PVS-Studio Tutorial. http://www.viva64.com/art-4-2-747004748.html